

PVS-Studio: analyzing ReactOS's code

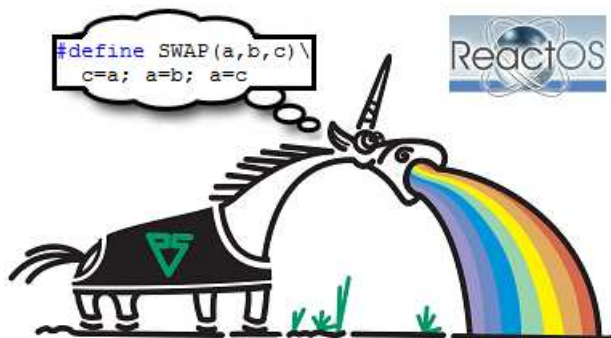
Author: Andrey Karpov

Date: 01.09.2011

Abstract

Having checked ReactOS's code I managed to fulfill three of my wishes at once. Firstly, I had wanted for a long time to write an article on a common project. It's not interesting to check the source code of projects like Chromium: its quality is too high and a lot of resources are spent to maintain it, which are unavailable to common projects. Secondly, it's a good example to demonstrate the necessity of static analysis in a large project, especially when it is developed by a diverse and distributed team. Thirdly, I've got a confirmation that PVS-Studio is becoming even better and more useful.

PVS-Studio is becoming better and better



I will start with the last point regarding the advantages of [PVS-Studio](#) tool. ReactOS indirectly confirms that PVS-Studio is developing in a right direction. Here is the news about checking ReactOS with such heavyweight as Coverity - "Coverity Redux" [[1](#)]. Of course, I understand that our tool's capabilities are far more modest than those of Coverity. However, PVS-Studio finds a whole lot of errors where Coverity has found "a few new errors". Besides, you are not forced to send the code anywhere; you can just pick up and check any project. It means we're on the right track.

What is ReactOS?

ReactOS is a contemporary, free and open-source operating system based on Windows XP/2003 architecture. The system was written from scratch and has the purpose of replicating the Windows-NT architecture created by Microsoft on all the layers from hardware to application layer. The size of the source code in C, C++ and Assembler is about 220 Mbytes.

References:

- [ReactOS Site](#).
- [Start Developing ReactOS](#).
- Wikipedia. [ReactOS](#).
- [ReactOS - Open-Source Windows Clone Software To Seriously Look Forward To](#).

Errors in ReactOS

Now let's speak about the whole lot of errors I have found in ReactOS's code. Of course, I won't describe them all in the article. [Here](#) I have laid out a text file with descriptions of errors found during analysis. The file contains diagnostic messages with file names and line numbers. I have also arranged the errors in a form of short code inserts and commented upon them. That's why those of you who would like to edit ReactOS should rely upon that file and not this article.

Or rather download PVS-Studio and check the project yourselves. You see, I'm not familiar with the project, so I copied out only those errors that I've understood. And regarding many fragments, I don't know if they contain errors or not. So my analysis is rather superficial. We will provide you a registration key if you want to check the project.

Errors you may come across in ReactOS are very diverse. It's a zoo of errors, really. There are misprints of one character.

```
BOOL WINAPI GetMenuItemInfoA(...)
{
    ...
    mii->cch = mii->cch;
    ...
}
```

This is how it should be actually written: "mii->cch = miiW->cch;". The letter 'W' was lost. As a result, applications can't trust the GetMenuItemInfoA function.

Here you are another misprint of one character. This time it's incorrect comparison of two names.

```
static void _Stl_loc_combine_names(_Locale_impl* L,
    const char* name1, const char* name2,
    locale::category c)
{
    if ((c & locale::all) == 0 || strcmp(name1, name1) == 0)
        ...
}
```

Operators && and & are mixed up. It's a very common error. I come across it virtually in every project where bits or file attributes are being handled.

```
static LRESULT APIENTRY ACeditSubclassProc()
{
    ...
    if ((This->options && ACO_AUTOSUGGEST) &&
```

```

        ((HWND)wParam != This->hwndListBox))

    ...

}

```

This is how the correct code must look: "(This->options & ACO_AUTOSUGGEST)". The sample below contains a similar error which causes the whole condition to be false all the time.

```

void adns__querysend_tcp(adns_query qu, struct timeval now) {

    ...

    if (!(errno == EAGAIN || EWOULDBLOCK || errno == EINTR ||
        errno == ENOSPC || errno == ENOBUFS || errno == ENOMEM)) {

        ...

    }
}

```

If you look closely, you may notice an insidious fragment: "|| EWOULDBLOCK ||".

By the way, in ReactOS I have found a lot of conditions which are always true or false. Some of them are not dangerous because, for instance, they are located in the assert() macro. But, in my opinion, there are some conditions which are crucial as well.

INT WSAAPI

```

connect(IN SOCKET s,

        IN CONST struct sockaddr *name,

        IN INT namelen)

{

    ...

    /* Check if error code was due to the host not being found */
    if ((Status == SOCKET_ERROR) &&

        (ErrorCode == WSAEHOSTUNREACH) &&

        (ErrorCode == WSAENETUNREACH))

    {

        ...

    }
}

```

You agree that the implementation of functions like "connect" should be tested as thoroughly as possible, don't you? But here we have a condition which is always false. It's not easy to notice the defect quickly, so let me explain the error:

```

(ErrorCode == 10065) && (ErrorCode == 10051)

```

By the way, the part relating to sockets looks very raw. Perhaps it is explained by the fact that it's an accepted practice to define SOCKET as a signed type in the Linux world, while in Windows it's unsigned:

```
typedef UINT_PTR SOCKET;
```

As a result, we have various errors in comparison operations:

```
void adns_finish(adns_state ads) {  
    ...  
    if (ads->tcpsocket >= 0) adns_socket_close(ads->tcpsocket);  
    ...  
}
```

The "ads->tcpsocket >= 0" expression is meaningless since it's always true.

There are simply odd fragments. Most likely, these are incomplete or forgotten code fragments.

```
if (ERROR_SUCCESS == hres)  
{  
    Names[count] = HeapAlloc(GetProcessHeap(), 0, strlenW(szValue) + 1);  
    if (Names[count])  
        strcmpW(Names[count], szValue);  
}
```

Why would you call the "strcmpW", if you will not use the result in any way?

There are errors in operations' priorities.

```
VOID NTAPI  
AtapiDmaInit(...)  
{  
    ...  
    ULONG treg = 0x54 + (dev < 3) ? (dev << 1) : 7;  
    ...  
}
```

I will add parentheses to show how this expression really works:

```
ULONG treg = (0x54 + (dev < 3)) ? (dev << 1) : 7;
```

The next error can always be found in any large project. There are a couple of these errors in ReactOS too. I mean the extra semicolon - ';'.
};

BOOLEAN

```
CTEScheduleEvent(PCTE_DELAYED_EVENT Event,  
                 PVOID Context)  
{  
    ...  
    if (!Event->Queued);  
    {  
        Event->Queued = TRUE;  
        Event->Context = Context;  
        ExQueueWorkItem(&Event->WorkItem, CriticalWorkQueue);  
    }  
    ...  
}
```

I am also fond of errors related to the initialization of array items. I don't know why. They are touching. Maybe it's just memories of my first experiments with arrays in Basic.

```
HPALETTE CardWindow::CreateCardPalette()  
{  
    ...  
    //include button text colours  
    cols[0] = RGB(0, 0, 0);  
    cols[1] = RGB(255, 255, 255);  
  
    //include the base background colour  
    cols[1] = crBackgnd;  
  
    //include the standard button colours...  
    cols[3] = CardButton::GetHighlight(crBackgnd);  
    cols[4] = CardButton::GetShadow(crBackgnd);  
    ...  
}
```

I may continue citing various interesting code fragments. Unfortunately, the article will become too long then so I have to stop. Let me remind you that you can read about the errors found in ReactOS in this [file](#). I will only cite the following piece of code for dessert:

```
#define SWAP(a,b,c)  c = a;\n                    a = b;\n                    a = c
```

An example of how it was used:

```
BOOL FASTCALL\nIntEngGradientFillTriangle(...)\n{\n    ...\n    SWAP(v2,v3,t);\n    ...\n}
```

This is a masterpiece.

Static code analysis

I find ReactOS a very good example of a project where regular static analysis is a mandatory necessity. The reason is not the developers' skill. It's because the project is very large and contains various subsystems. It means that there are always a lot of people working on such a project. And in a large team there are always people whose programming skill is relatively worse or better; some programmers use one style and others use another style. But nobody is safe from errors. Look at the following code.

This is just what one person had written in ReactOS:

```
if ((res = setsockopt(...)) == -1))
```

The code doesn't work as it was intended. The correct code is the following: `if ((res = setsockopt(...)) == -1)`. If you adhere to practice of always writing a constant in the beginning, you will never make a wrong assignment inside the "if" operator. We have a different sort of error here. But if you follow the rule above when writing the code, then you won't make a mistake in the expression at hand as well: `"if (-1 == res = setsockopt(...))"`.

But even if you follow that practice, you can easily make a mistake in an alternative way.

```
static DWORD CALLBACK\nRegistrationProc(LPVOID Parameter)\n{\n    ...
```

```

    if (0 == LoadStringW(hDllInstance, IDS_UNKNOWN_ERROR,
                        UnknownError,
                        sizeof(UnknownError) /
                        sizeof(UnknownError[0] - 20)))
        ...
}

```

The 0 constant is written nicely here. But the closing parenthesis is in a wrong place. It's a simple misprint.

What for do I cite all these examples? To show you that no one of us programmers is ideal. Neither coding standards, nor programming technologies, nor self-discipline guarantee that you won't make mistakes in source code.

In large projects you just cannot do without auxiliary technologies like dynamic and static analysis. I want to emphasize the following idea:

I believe that static code analysis should be a mandatory component of the development cycle in the case of ReactOS and other large projects.

Let me explain my statement. In such systems, you cannot get close to 100% code coverage when testing the code with unit-tests or regression tests. Well, to be more precise, you can, of course, but costs of creating and maintaining such tests will become unacceptably high.

The reason is that the number of system's possible states and execution paths of code branches is too large. Some branches get control rarely but they do not get less important because of that. It is here that you can notice the advantage of static analysis. It checks the whole source code regardless of how often it gets control during the program's execution.

Here is an example of checking a code that rarely gets control:

```

static HRESULT STDMETHODCALLTYPE
CBindStatusCallback_OnProgress(...)
{
    ...
    if (This->szMimeType[0] != _T('\0'))
        _tprintf(_T("Length: %I64u [%s]\n"), This->Size,
                This->szMimeType);
    else
        _tprintf(_T("Length: %u11\n"), This->Size);
    ...
}

```

```
}
```

It's most likely that the code was written incorrectly in the beginning. Then somebody noticed that the message was generated in a wrong way and fixed it by writing "%l64u". But he paid no attention to the code nearby, while it still has an incorrect format "%ull". This brunch seems to be called very rare. Static analysis won't miss that. It hadn't actually, since I can show you this example.

Another good example is a large number of memory cleanup errors that I've found in ReactOS. I understand why there are so many of them. Nobody checks whether the memory is filled or not. Firstly, it's difficult to realize that you might make a mistake in such simple places. Secondly, it's not so easy to verify if some temporary buffer in a function has been cleared or not. Static analysis again comes to your aid here. Let me give you only a couple of examples. Virtually I have counted at least 13 errors of filling arrays with a constant value.

```
#define MEMSET_BZERO(p,l) memset((p), 0, (l))
```

```
char *SHA384_End(SHA384_CTX* context, char buffer[]) {  
    ...  
    MEMSET_BZERO(context, sizeof(context));  
    ...  
}
```

Only the first bytes of the array are cleared, since `sizeof(context)` returns the pointer's size instead of the structure's size.

```
#define RtlFillMemory(Destination, Length, Fill) \  
    memset(Destination, Fill, Length)
```

```
#define IOPM_FULL_SIZE          8196
```

```
HalpRestoreIopm(VOID)  
{  
    ...  
    RtlFillMemory(HalpSavedIoMap, 0xFF, IOPM_FULL_SIZE);  
    ...  
}
```

Arguments are mixed up when using the `RtlFillMemory` macro. This is how the call should look:

```
RtlFillMemory(HalpSavedIoMap, IOPM_FULL_SIZE, 0xFF);
```


To tabs and spaces again

I want to ask you beforehand not to start a flame on the topic in comments. I will simply tell you my opinion. You may agree with it or not, but let's not discuss it.

There are two irreconcilable camps. One of them stands for using tabs in code because it allows you to adjust code presentation according to your preferences. The others say that it doesn't work anyway and there are no good reasons for using tabs. Tabs cause only harm and spoiled formatting. I refer to the latter camp.

We may eternally repeat that everything will be okay if tabs are used in a right way. Unfortunately, people who say that work on one project in isolation, without interacting with the outer world. In any open-source or simply large project you cannot obtain a good code formatting if it is permitted to use tabulation of any kind.

I won't get involved into abstract discussions. This time I will simply cite an obvious example from ReactOS's code to my opponents.

ReactOS's coding standard has a good rule from the theoretical viewpoint [\[2\]](#):

Generic note about TABs usage: Don't use TABs for formatting; use TABs for indenting only and use only spaces for formatting.

Example:

```
NTSTATUS
```

```
SomeApi(IN Type Param1,  
[spaces]IN Type Param2)  
{  
[TAB]ULONG MyVar;  
[TAB]MyVar = 0;  
[TAB]if ((MyVar == 3) &&  
[TAB][sp](Param1 == TRUE))  
[TAB]{  
[TAB][TAB]CallSomeFunc();  
...  
}
```

TAB fans are satisfied. But I open up ReactOS's sources and observe spoiled formatting in many places. Why is that?

```

→   ....break;
→   case WM_KILLFOCUS:
→   |   ....if ( (This->options && ACO_AUTOSUGG
→   →   ( (HWND) wParam != This->hwndListBox) )
→   ....{
→   →   ShowWindow(This->hwndListBox, SW_HIDE)
→   ....}
→   ....break;

```

```

→   case VT_UI1:
→   ....if (readit) {
→   →   DWORD x;
→   →   hres = xbuf_get(buf, (LPBYTE) &x, size
→   →   if (hres) ERR("Failed to read integ
→   →   memcpy(arg, &x, 1);
→   ....}
→   ....if (debugout) TRACE_(olerelay) ("%
→   ....return hres;
→   case VT_BSTR: {
→   ....if (readit)
→   ....{
→   →   ULONG flags = MAKELONG(MSHCTX_DIFFE

```

```

→   ....
→   ....if(xmlC14NStrEqual(prefix, (ns1·
→   →   if(xmlC14NStrEqual(href, (ns1·!=·N
→   →   |   ....return(xmlC14NIsVisible(ct
→   →   } else {
→   →   ....return(0);
→   →   }
→   ....}

```

The answer is obvious. Because it's hard to remember where you should press TAB and where you should press several spaces when the project is not the only one you are dealing with. That's why people constantly make mistakes. Since it comes to that, let's be practitioners, not theorists. Why not forbid usage of tabs at all? Then we all will write code with the same formatting and if a violator appears who starts using tabs, it will be easy to find and reprimand him.

It's not a step backward in code formatting! It's just a step forward! It's the next level of awareness. Theoretical beauty of indenting does not match practice. First of all, it's important to provide unequivocal code representation and easy development process in a large team. The Google company understands that. Their formatting standard uses only spaces [\[3\]](#). Those who stand for using tabs, please

think it over why it is spaces that a distributed team of highly skilled professionals working on Chromium has chosen for formatting.

And once again, the theoretical beauty of configurable indenting does not match practice. However nice the theory sounds, it's of no use if it doesn't work. And this is how things are in ReactOS.

So my recommendation to the ReactOS development team is to modify their standard and to refuse usage of tabulation. Any tab should be considered a mistake and eliminated from the code.

By the way, this practice will allow you to detect awful things like the following one in ReactOS's code:

BOOLEAN

```
KdInitSystem(IN ULONG BootPhase,
             IN PLOADER_PARAMETER_BLOCK LoaderBlock)
{
    ...
    /* Check if this is a comma, a space or a tab */
    if ((*DebugOptionEnd == ',') ||
        (*DebugOptionEnd == ' ') ||
        (*DebugOptionEnd == '\t'))
        ...
}
```

The last comparison is comparison to a tab, not a space, as it may seem. The right code must be the following: `("*DebugOptionEnd == '\t')`.

Note for TAB fans. Please, don't tell me again how to use tabs in a right way. And this is not my code. Look, there is a concrete project like ReactOS. It has a badly formatted code. Now think how to save a new programmer opening the project's code from making guesses about what TAB size should be set in the editor's settings. Ideas like "they should have written it right from the beginning" are of no practical value.

References

1. Newsletter 79. Coverity Redux. <http://www.viva64.com/go.php?url=727>
2. ReactOS. Coding Style. <http://www.viva64.com/go.php?url=724>
3. Google C++ Style Guide. <http://www.viva64.com/go.php?url=679>